Just-In-Time Data Virtualization: Lightweight Data Management with ViDa

Manos Karpathiotakis[†] Ioannis Alagiannis[†] Thomas Heinis[†‡] * Miguel Branco[†]
Anastasia Ailamaki[†]

[†]Ecole Polytechnique Fédérale de Lausanne
{firstname.lastname}@epfl.ch

[‡]Imperial College
t.heinis@imperial.ac.uk

ABSTRACT

As the size of data and its heterogeneity increase, traditional database system architecture becomes an obstacle to data analysis. Integrating and ingesting (loading) data into databases is quickly becoming a bottleneck in face of massive data as well as increasingly heterogeneous data formats. Still, state-of-the-art approaches typically rely on copying and transforming data into one (or few) repositories. Queries, on the other hand, are often ad-hoc and supported by pre-cooked operators which are not adaptive enough to optimize access to data. As data formats and queries increasingly vary, there is a need to depart from the current status quo of static query processing primitives and build dynamic, fully adaptive architectures.

We build ViDa, a system which reads data in its raw format and processes queries using adaptive, just-in-time operators. Our key insight is use of virtualization, i.e., abstracting data and manipulating it regardless of its original format, and dynamic generation of operators. ViDa's query engine is generated just-in-time; its caches and its query operators adapt to the current query and the workload, while also treating raw datasets as its native storage structures. Finally, ViDa features a language expressive enough to support heterogeneous data models, and to which existing languages can be translated. Users therefore have the power to choose the language best suited for an analysis.

1. INTRODUCTION

Whether in business or in science, the driver of many big data applications is the need for analyzing vast amounts of heterogeneous data to develop new insights. Examples include analyzing medical data to improve diagnosis and treatment, scrutinizing workflow data to understand and optimize business processes, analyzing stock market tickers to support financial trading, etc. Yet, as different as these examples are, their core challenges revolve around providing unified access to data from heterogeneous sources, which remains a formidable challenge today.

State-of-the-art approaches have relied on data integration technologies to place all data, originally stored in heterogeneous file formats located at different sources, in one data warehouse. In this process, semantic integration approaches [20] help to map semantically equivalent data from different data sources on a common schema. Physical integration, on the other hand, is commonly addressed by first transforming all heterogeneous data into a common format and then copying and integrating it into a data warehouse. Transforming and integrating all data into a warehouse, however, is no longer an option for a growing number of applications. For example, in many scenarios institutions owning the data want to retain full control for legal or ethical reasons. Transforming and loading the data into a warehouse is a considerable time investment that is unlikely to pay off as not all data may be accessed, while it bears the risk of vendor lock-in. Statically transforming all data into one common format may even impede query execution, as different query classes require data to be organized into different layouts for efficient query processing. Finally, for applications facing high update rates, preserving data freshness requires a continuous synchronization effort to propagate updates on the original data sources to the data warehouse in a timely manner.

A change of paradigm is required for data analysis processes to leverage the amounts of diverse data available. Database systems must become dynamic entities whose construction is lightweight and fully adaptable to the datasets and the queries. *Data virtualization*, i.e, abstracting data out of its form and manipulating it regardless of the way it is stored or structured, is a promising step in the right direction. To offer unconditional data virtualization, however, a database system must abolish static decisions like pre-loading data and using "pre-cooked" query operators. This dynamic nature must also be extended to users, who must be able to express data analysis processes in a query language of their choice.

1.1 Motivating Applications

One of the key visions and driver of the Human Brain project (HBP [37]) is to improve diagnosis and treatment of brain related diseases. Defining sound disease characterizations of brain diseases shared by patients is a necessary first step and requires a representative and large enough sample of patients. Researchers in the HBP consequently must access and analyze data from hospitals across Europe. En-

*Work done while the author was at EPFL.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

abling access to heterogeneous data at different hospitals, however, is a massive integration challenge.

Integrating all patient data into one warehouse, i.e., transforming it physically and semantically into the same representation and moving it into one administrative location, seems to be the most straightforward approach to enable its analysis. Nevertheless, given the heterogeneous data formats (results from various instruments or processing pipelines are stored as JSON, CSV, medical image formats containing arrays, etc.), the frequent updates to medical records and the different analysis queries from aggregates to graph queries, importing all data into a warehouse is impractical. More importantly, national legislation and institutional ethics guidelines make any approach where patient data is moved, copied or transformed impossible. Instead, data must remain at the hospitals. In a nutshell, the major data management challenge lies in optimizing the physical integration of data stored in heterogeneous formats to efficiently support heterogeneous queries.

Similar challenges are met in other applications as well. Banks, for example, operate large numbers of databases and data processing frameworks. A single data access layer managed by different functional domains (e.g., Trading, Risk, Settlement) is needed, but none is available. Existing integration systems are impractical to use at this scale, and regulations also require banks to keep raw data and correlate it directly with the trade life cycle. Accessing all data in its raw form, on the other hand, allows different functional domains in banks to easily interface with the data from others without having to share a common system, and independently of data models or formats. This form of "ad hoc" data integration allows different communities to create separate databases, each reflecting a different view/area of interest over the same data.

1.2 The ViDa Approach

To address the challenges of the aforementioned use cases as well as many other examples stemming from today's and future applications, we clearly have to move beyond the state of the art. Data management must become a lightweight, flexible service, instead of a monolithic software centering around the status quo of static operators and growing obese under the weight of new requirements. This paper describes ViDa, a novel data management paradigm offering *just-in-time data virtualization* capabilities over raw data sources.

ViDa envisions transforming databases into "virtual" instances of the raw data, with users spawning new instances as needed where they can access, manipulate and exchange data independently of its physical location, representation or resources used. Performance optimizations are transparent, entirely query-driven and performed autonomously during query execution. Constant changes to a database, including schema changes, are not only supported but encouraged so that users can better calibrate their "view" over the underlying data to their needs. Data analysts build databases by launching queries, instead of building databases to launch queries. The database system becomes naturally suited for data integration. Figure 1 depicts a business intelligence architecture based on data virtualization. A transparent virtualization layer encompasses all data and enables efficient data access. Data processing frameworks use the virtualization layer to provide fast end-user services and seamlessly share data access optimizations.

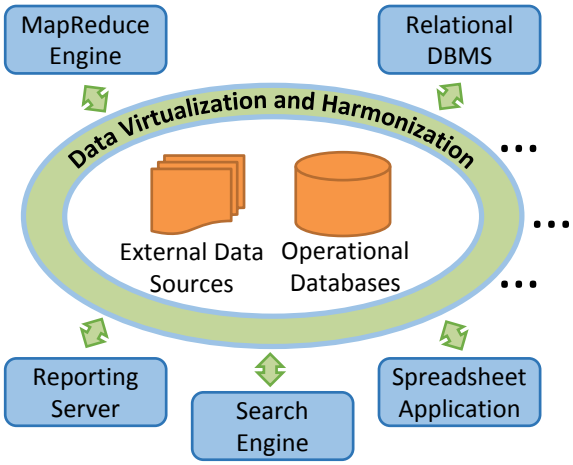

Figure 1: A Business Intelligence Architecture based on data virtualization.

Working towards this vision, ViDa leverages efforts to abolish the static nature of DBMSs. To efficiently support diverse data models like tables, hierarchies and arrays, ViDa employs code generation [4, 31, 41] to adapt its engine's operators and internal data layout to the models of the data to be processed. In addition, ViDa operates over raw datasets to avoid data loading. To also avoid undue performance penalties related to raw data accesses [3, 28], it generates access paths aggressively optimized [31] for the formats data is stored in. Finally, we develop a query language for ViDa that enables formulating queries over data organized in heterogeneous models and is expressive enough for popular query languages to be translated to it. ViDa thus decouples language from data layout, and enables users to use their language of preference.

As a proof of concept, we use a preliminary prototype of ViDa to run a workload from the Human Brain Project, and show that data virtualization using ViDa is a viable option for data analysis.

Paper organization. The remainder of this paper is structured as follows. We outline the architecture of ViDa in Section 2. Section 3 presents the query language that ViDa uses. In Section 4 we present the just-in-time query executor of ViDa. Section 5 discusses optimization opportunities, and in Section 6 we confirm the design decisions with a set of experiments. Section 7 discusses the next steps in our effort towards data virtualization. Finally, Section 8 discusses related work, before we conclude in Section 9.

2. BUILDING DATABASES JUST IN TIME

Key to ViDa is that data is left in its raw form and acts as the major "golden data repository". By doing so, we do not need to make static decisions about data organization, representation and optimization through building a database, but instead can defer decisions to runtime when data, and more importantly, the queries are known. Analysis of data in disparate sources thus begins with ad hoc querying and not by building a database, or put differently, data analysts build databases *just-in-time* by launching queries as opposed to building databases to launch queries.

To implement this vision, we need to define and implement i) an internal data representation to describe heterogeneous raw datasets, and a query language expressive enough to combine heterogeneous data sources and able to be translated into popular query languages, ii) a query processor that creates its operators just-in-time based on the query requirements and the underlying datasets, and iii) an optimizer which considers ViDa's code generation abilities and takes into account that ViDa natively operates over raw data.

Representing and querying raw data. Data sources are typically stored in various formats (e.g., CSV, XML, JSON, existing DBMS), each potentially based on a different data model. ViDa therefore provides a grammar to concisely describe heterogeneous datasets, and a query language which is general enough to cope with the data model heterogeneity and which supports arbitrary data transformations. Crucially, ViDa's query language is generic enough for other query languages to be translated to it, thus enabling data analysts to use the best suited query language (e.g., SQL, XQuery) for their data analysis. Depending on the query and the language used, ViDa internally uses different representations of the data. For example, computing the result of an aggregation function (e.g., max) can be naturally expressed in SQL, and thus benefits from a tabular representation of the data. On the contrary, a graph representation makes computing the shortest path between two nodes a natural operation.

A just-in-time query executor. Classical DBMS are highly optimized, but their monolithic query engines achieve efficiency at the expense of flexibility. On the contrary, ViDa aims at preserving flexibility on the supported formats while also ensuring efficient query processing regardless of the input format. ViDa achieves this by adapting to the queries and to the underlying raw datasets. The query executor of ViDa applies code generation techniques to dynamically craft its internal operators and its access paths (i.e., its scan operators) at runtime. These operators provide efficient access to external data by generating optimized access and processing methods for different data formats and models.

Optimization for raw data querying. Classical query optimizers do not capture the universe of decisions that are available to queries asked over a wide variety of raw heterogeneous datasets. For example, textual tabular files require different handling than relational tables that contain the same information. The dynamic nature of ViDa is also an enabler for novel optimizations, each of them potentially targeting specific categories of raw data formats. ViDa thus features an optimizer which takes into account that queries are executed on raw data, and which is aware that ViDa's code generation capabilities enable proposing query plans very specific to queries.

2.1 ViDa Architecture

Figure 2 illustrates a high-level description of ViDa. Incoming queries are translated to the internal "wrapping" query language, or expressed directly in it to enable accesses across data models. The entire query execution phase is monitored by ViDa's optimizer, which extends classical optimizers with runtime decisions related to raw data accesses. The optimizer is responsible for performing the query rewriting and the conversion of a logical to a physical query plan. In the physical query plan, the various abstraction layers of the database engine collapse into highly efficient machine

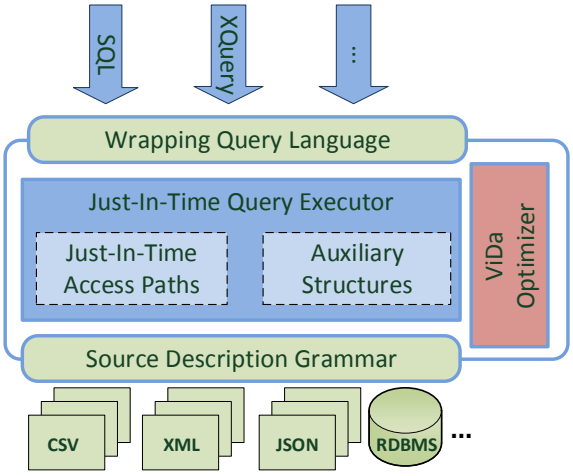

Figure 2: The architectural components of ViDa.

code for the current query. Each dataset's description is used to adapt the generated code to the underlying formats and schemas. The "capabilities" exposed by each underlying data source dictate the efficiency of the generated code. For example, in the case that ViDa treats a conventional DBMS as a data source, ViDa's access paths can utilize existing indexes to speed-up queries to this data source.

When applicable, as when handling CSV and JSON files, the generated operators of ViDa use auxiliary data structures specifically designed to reduce raw data access costs [3] (e.g., parsing). ViDa also maintains caches of previously accessed data. As ViDa targets analytical scenarios, the workloads we deal with are typically read-only or append-like (i.e., more data files are exposed), therefore maintenance or extension of ViDa's data structures is straightforward. ViDa currently handles the cases of in-place updates transparently. Updates to the underlying files result in dropping the auxiliary structures affected.

The following sections present how ViDa enables querying a wide spectrum of raw data via an expressive query language, a query engine that is generated just-in-time in its entirety, and an optimizer defining the appropriate engine structure.

3. MULTIPLE MODELS, ONE LANGUAGE

To execute queries on structured raw datasets, ViDa requires an elementary description of each data format. The equivalent concept in a DBMS is a catalog containing the schema of each table. ViDa feeds this information to the query engine, so that it generates the access paths at runtime to read each data source. ViDa also requires an expressive query language to support the underlying heterogeneous data models, which can also be translated into the user's preferred language for data analysis. This section discusses the raw data descriptions and the query language we develop.

3.1 Raw Data Descriptions

Data files follow a variety of formats. Typical examples are CSV, XML, JSON, but there is also a variety of ad hoc and of domain-specific formats; ROOT [7] for high-energy physics, FITS [49] for astronomy, etc. While each format's specifics vary widely (e.g., character encoding, headers, con-

ventions, etc.), most formats follow a defined underlying structure. CSV files represent tables or vertical/horizontal partitions of tables. FITS files contain tables in either ASCII or binary encoding, in addition to arrays. XML and JSON files contain semi-structured data. In the context of ViDa, we identify widely-used data formats and define a minimal grammar that is sufficiently rich to describe the structure of the raw data to the upper layers of the query engine. To support arbitrary data formats with unknown a priori schemas, we design ViDa flexible enough to support additional formats if their description can be obtained through schema learning tools [56].

ViDa uses data source descriptions that capture i) the *schema* of each raw dataset, ii) the *"unit"* of data retrieved using the data access interface for each dataset format, and iii) the *access paths* exposed for each dataset format. This information is required to validate user queries, but also to be fed to the query engine so that it adapts to each underlying dataset instance.

Suppose a file contains data modeled as a matrix. Array data is common in formats such as ROOT and NetCDF. A simplified description of the file's schema could be:

```
Array(Dim(i,int), Dim(j,int), Att(val))
val = Record(Att(elevation, float),
             Att(temperature, float))
```

If we rely on a straightforward parser to access our example array file, the "unit" read with each data access will probably be a single element - an *(elevation,temperature)* pair. If, however, a more sophisticated parser is available, or if the data format in question is accompanied by an extensive data access framework (as is the case with scientific data formats such as HDF5), alternative "units" of data can be i) a row of the matrix, ii) a column of the matrix, or even iii) a $n \times m$ chunk of the matrix (as is also more common in array databases [5, 47]). In a database system, this datum is usually a page with tuples (for a row-store), or a piece of a column (for a column-store). For other data formats, other "units" are possible, such as objects in the case of JSON. Finally, the access paths exposed by a file include, among others, sequential scans, index-based accesses, and accesses based on an identifier (e.g., a rowId).

3.2 A Query Language for Raw Data

Queries targeting raw data must consider the unavoidable model heterogeneity, and enable the combination of information from diverse data sources. The language used must also enable users to "virtualize" the original data, i.e., apply powerful transformations over the output of a query. ViDa uses a query language that provides support for a multitude of data models. Specifically, collection types such as sets, bags, lists, and multi-dimensional arrays are supported. This flexibility enables queries to transparently access a great variety of datasets (e.g., relational tables, CSV and JSON files, array image data, etc.).

Existing work in the area of raw data querying focuses on providing support for a single data model [3, 6], or maps non-conforming datasets to the relational model [31]. As ViDa aims for native support of non-relational data sources, however, the relational calculus is not sufficient as a base for its query language. ViDa therefore uses the *monoid comprehension calculus*, detailed in [22, 23].

Monoid comprehension calculus. A *monoid* is an algebraic construct term stemming from category theory.

NULL	null value
c	constant
v	variable
$e.A$	record projection
$\langle A_1 = e_1, ..., A_n = e_n \rangle$	record construction
if e_1 **then** e_2 **else** e_3	if-then-else statement
e_1 op e_2	op: primitive binary function (e.g., $<, +$)
$\lambda v : \tau.e$	function abstraction
$e_1(e_2)$	function application
$\mathcal{Z}_\oplus$	zero element
$\mathcal{U}_\oplus(e)$	singleton construction
$e_1 \oplus e_2$	merging
$\oplus\{e \mid q_1, ..., q_n\}$	comprehension

Table 1: The monoid comprehension calculus

A monoid of type T comprises an associative binary operation $\oplus$ and a zero element $\mathcal{Z}_\oplus$. The binary operation, called *merge function*, indicates how two objects of type T can be combined. The *zero element* $\mathcal{Z}_\oplus$ is the left and right identity of the merge function $\oplus$; for every object x of type T, the equivalence $\mathcal{Z}_\oplus \oplus x = x \oplus \mathcal{Z}_\oplus = x$ is satisfied.

Monoids can be used to capture operations between both primitive and collection data types. The latter also require the definition of a *unit function* $\mathcal{U}_\oplus$, which is used to construct singleton values of a collection type (e.g., a list of one element). For example, $(+, 0)$ represents the primitive *sum* monoid for integer numbers. The pair $(\cup, \{\})$ along with the unit function $x \rightarrow \{x\}$ represent the *set* collection monoid.

The *monoid comprehension calculus* is used to describe operations between monoids. A monoid comprehension is an expression of the form $\oplus\{e|q_1, ..., q_n\}$. The terms q_i are called *qualifiers*. Each qualifier can either be

- a *generator*, taking the form $v \leftarrow e'$, where e' is an expression producing a collection, and v is a variable that is sequentially bound to each value of said collection.
- a *filter* predicate.

The expression e is called the *head* of the comprehension, and is evaluated for each value binding produced by the generators. The evaluation results are combined using the merge function $\oplus$, called the *accumulator* of the comprehension in this context. Table 1, originally from [23], contains the syntactic forms of the monoid comprehension calculus.

The comprehension syntax that ViDa uses is slightly altered but equivalent to the one presented, and resembles the sequence comprehensions of Scala. The syntax we use is $for\{q_1, ..., q_n\}\ yield \oplus e$. As an example, suppose that we want to pose the following SQL query counting a department's employees:

```
SELECT COUNT(e.id)
FROM Employees e JOIN Departments d ON (e.deptNo = d.id)
WHERE d.deptName = "HR"
```

The same aggregate query can be expressed in our version of monoid comprehension syntax as:

```
for { e <- Employees, d <- Departments,
      e.deptNo = d.id, d.deptName = "HR"} yield sum 1
```

This example requires us to use the *sum* monoid for integers to perform the count required. Other primitive monoids

we can use in our queries include, among others, *max*, *average*, and *median*. Similarly, queries with universal or existential quantifiers can use the $\vee$ and $\wedge$ monoids for boolean types. More complex monoids include the *ordering* monoid, the *top-k* monoid, etc.

Monoid comprehensions also support nested expressions. A query requesting an employee's name along with a collection of all the departments this employee is associated with is expressed as:

```
for { e <- Employees, d <- Departments, e.deptNo = d.id}
yield set (emp := e.name,
           depList := for {d2 <- Departments,d.id = d2.id}
                      yield set d2)
```

Expressive Power. Monoid comprehensions bear similarities to list and monad comprehensions [9, 53], which are constructs popular in functional programming languages. We opt for monoid comprehensions as a "wrapping" layer for a variety of languages because they allow inputs of different types (e.g., sets and arrays) to be used in the same query. Query results can also be "virtualized" to the layout/ collection type requested; different applications may require different representations for the same results. Examples include, among others, representing the same original data either as a matrix or by using a relational-like tabular representation, and exporting results as bag collections while the original inputs are lists. This capability aids in "virtualizing" the original data as per the user's needs.

Crucially, ViDa's language based on monoid comprehensions lends itself perfectly to translation to other languages. Support for a variety of query languages can be provided through a "syntactic sugar" translation layer, which maps queries written in the original language to the internal notation. Doing so enables users to formulate queries in their language of choice. Specifically, monoid comprehensions are the theoretical model behind XQuery's FLWOR expressions, and also an intermediate form for the translation of OQL [23]. The monoid comprehension calculus is also sufficient to express relational SQL queries. SPARQL queries over data representing graphs can also be mapped to the monoid comprehensions calculus [13].

The prototype version of ViDa presented here uses comprehensions as its query interface; the syntactic sugar translation layers are ongoing work. During query translation, ViDa translates the monoid calculus to an intermediate algebraic representation [23], which is more amenable to traditional optimization techniques. ViDa's executor and optimizer operate over this algebraic form.

ViDa is the first approach applying monoid comprehensions to enable virtualization of raw heterogeneous datasets, and coupling such an expressive calculus with the application of code generation techniques in all steps of query evaluation to achieve our vision of just-in-time databases. Generating a query engine just-in-time is connected to the inherent complexity of evaluating queries formulated in such an expressive language and to the large universe of decisions related to accessing raw data.

4. DATABASE AS A QUERY

ViDa is designed on the premise that a database must be a dynamic entity whose construction is lightweight and fully adaptable to the queries and the underlying data instances, rather than the end product built by a laborious process and then exploited for answers. The query engine of ViDa therefore employs code generation techniques to produce at runtime code specific for each incoming query and for each underlying raw dataset accessed. The rest of the section elaborates on the challenges that static designs face, before contrasting them with the just-in-time executor of ViDa.

Adapting an engine to underlying data. Traditional database engines are designed to handle hard-coded data structures. In a row-store, this structure is the database "page"; in a column-store, it is typically a tightly-packed array of data containing a single column of a table. Both these variations, however, presuppose that data has been transformed to the appropriate binary format, and then loaded into a database. Reformatting a user's raw data into new structures managed by the database engine and loading it is a costly process [31]. Statically defining these structures is also not beneficial, because given the variety in data models and formats that ViDa combines, different queries may benefit from a different choice of data structure layout. Even when dealing with strictly relational data, query performance improves if the layout is able to adapt to the workload needs [4]. Considering the diverse and arbitrary nature of workloads that can be executed over a collection of heterogeneous datasets, from simple SQL aggregations to combinations of JSON with array data, it becomes apparent that the static decisions inherent to traditional database systems must be replaced with dynamic ones.

An engine with fine-grained operators. The evaluation process for a random query mapped to the monoid comprehension syntax, as detailed in [23], introduces overheads if executed in a static manner. An incoming query to be evaluated is initially translated to comprehension syntax. After applying a series of rewrite rules to optimize the query (e.g., remove intermediate variables, simplify boolean expressions, etc.) the partially optimized query is translated to a form of nested relational algebra which is much closer to an execution plan, and over which an additional number of rewritings can be applied (e.g., unnesting sub-queries that could not be unnested while in comprehension syntax).

Operators of this form, however, are much more complex than their relational counterparts. For example, our algebra includes the *reduce* operator, which is is a generalization of the straightforward relational projection operator. Besides projecting a candidate result, it optionally evaluates a binary predicate over it. The operator's behavior also changes depending on the type of collection to be returned (e.g., no duplicate elimination is required if the result type is a bag). Static implementations of such operators have to be very generic to handle all their functionality requirements.

Another concern is that in ViDa the layout of in-memory data structures may vary (e.g., different layouts may be used for data stored in a different - or even the same - file type if it benefits execution). Finally, even if the intermediate results fed to the operator are tabular, the user may have requested the output transformed as an arbitrarily nested object. A "pre-cooked" operator offering all these capabilities must be very generic, thus introducing significant interpretation overhead [41].

4.1 Just-in-time Operators

ViDa generates its internal structures and query operators on demand, to better suit the workload needs. Decisions concerning query execution and operator structure are

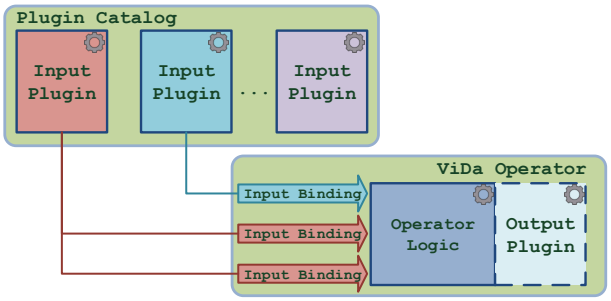

Figure 3: High-level layout of a ViDa JIT query operator.

postponed until runtime, when as much useful information for the query invocation as possible has been gathered. ViDa generates its query operators just-in-time, and is thus able to flush out optimal, special-purpose operators satisfying the needs of the currently processed query. A typical ViDa operator, depicted in Figure 3, is created in three steps.

Input Plugins. As its first step, the operator needs to process its inputs, which it obtains either by directly scanning a raw file (if it is a *scan* operator) or by having a previous operator pipeline its output to the current one. All ViDa operators can accept input expressed in a variety of data models and formats. Every operator invokes a file-format-specific input plugin for each of its input bindings to be able to process them. Multiple bindings may utilize the same plugin; for example, two bindings may originate from CSV files, so a CSV plugin is used for both.

The more domain-specific knowledge we have for a raw dataset format to be accessed by ViDa and the queries to be run over it, the more efficient its corresponding plugin is. For example, an organization may use as logs JSON files that do not contain arrays. ViDa can use a specialized JSON plugin for these files, reducing CPU overheads, and a general plugin for generic JSON files.

Input plugins also use auxiliary structures to speed-up accesses to raw data files. For textual data formats (CSV and JSON), auxiliary structures which index the *structure* of raw data [3, 43] are used to support efficient navigation in the text files and reduce the parsing costs incurred to identify fields of interest. Other file formats (e.g., HDF [50]) contain indexes over their contents; these indexes can also be used by the scan operators of ViDa to speed-up accesses.

Operator Logic. By calling the appropriate plugin for the data instance in question, custom code is generated to read a "datum" of the input datasets (e.g., a JSON object, or even a pre-materialized tuple in a binary format). Once the generated code places the necessary data elements in ViDa's data caches (or ViDa confirms that the data elements are already in its caches), the code corresponding to the exact operator logic for this query instance is generated. Before launching query execution, the optimizer of ViDa has taken into account combinations of data already cached, potentially in different layouts (e.g., chunks of an array against a column from a relational-like CSV table), to decide on the operator design to generate. The code that is generated depends on the data layout to be used for this operator invocation. The generated code is also stripped from general-purpose checks that introduce overheads (e.g., branches whose outcome is known at runtime, such as datatype checks).

In the general case, the operators of ViDa attempt to pipeline data when possible. Pipelined execution, combined with the fact that the query executor generates low-level machine code, enables ViDa to effectively operate directly over raw data. The access paths of ViDa (i.e., the scan operators) do not materialize any intermediate data structures at an early stage; no "database page" or "data column" has to be built to answer a query. Instead, data bindings retrieved from each "tuple" of a raw file are placed in CPU registers and are kept there for the majority of a query's processing steps [41]. Of course, some implementations of operators such as joins typically are materializing, but data structure creation costs are still reduced.

In certain scenarios, such as when ViDa incurs I/O (or predicts that it will), we use more conservative strategies: The scan operators of ViDa eagerly populate data structures, especially if part of the data structure population cost can be hidden by the I/O cost of the initial accesses to the raw data files; pipelined execution is then used for the rest of the query tree.

Operator Output. If an operator has to explicitly materialize part of its output (e.g., relational joins and aggregations are blocking operators, and typically populate a hashtable internally), an output plugin is called to generate the code materializing the data in an appropriate layout and format. The output format depends on various factors, including: i) the compactness of the format (e.g., binary JSON serializations are more compact than JSON), ii) the requested output format of the data to be projected as the last step of the query (e.g., the user may require the output in CSV), and iii) potential locality in the workload (e.g., intermediate results of the current query can be reused by future queries). Section 5 presents additional information on these decisions.

5. OPTIMIZATION FOR JIT DATABASES

Applying traditional optimization techniques to ViDa is essential to exploit the decades of research in query optimization for database engines. Accessing raw data, however, introduces trade-offs, both implicit and explicit, that ViDa must take into account during the query optimization process. This section discusses the challenges related to applying traditional optimization approaches to ViDa. ViDa's dynamic nature also enables making, and potentially correcting, optimization decisions just-in-time and we thus discuss new opportunities for optimization as well.

Optimizing Just-in-time. When considering optimization decisions, ViDa differs from a traditional database system, mostly because of its capability to generate ad hoc operators. To reduce the cost of raw data accesses, ViDa considers information such as i) the incoming query needs (e.g., output format, projectivity, etc.), ii) the schema of the underlying raw data, iii) the auxiliary structures that have been built for the raw data, and iv) the format and layout of data currently in caches. Besides the standard optimization phase that is similar to that of classical optimizers, ViDa has to make fine-grained decisions which affect each operator's generation. For example, it is up to the optimizer to decide what data layout each query operator should use during the evaluation of an incoming query. The optimizer of ViDa makes some of these decisions statically, using heuristics and statistics that have been gathered. The desired behavior of some operators, however, may be specified just-in-time, so

it is non-trivial to fully model the entire execution statically. Therefore, at runtime ViDa both makes some decisions and may change some of the initial ones based on feedback it receives during query execution [17]. In this case, ViDa generates the code for a new instance of its engine on the fly based on the modified query plan.

Perils of Classical Optimization on Raw Data. Traditionally, operators that read data from a relation which resides in the buffer pool of a DBMS have to "project" the attributes of interest to the query. The CPU cost per attribute fetched is on average the same. For operators accessing raw data, however, the cost per attribute fetched may vary between attributes due to the effort needed to navigate in the file. For instance, for CSV files, reading new attributes requires tokenization, parsing, and conversion of data. Therefore, the CPU costs are generally higher and more varied than in a traditional system, where reading a "tuple" (or a subset of it) from the buffer pool is a constant factor for each scan operator. The more complex the format, the more expensive this process usually is.

Unfortunately, there is no single formula to model these access costs. For textual data formats (CSV and JSON), auxiliary structures which index the *structure* of the underlying raw data have been proposed [3, 43]. For a CSV file, binary positions of a file's fields are stored in a positional index during initial accesses, and are used to facilitate navigation in the file for later queries. The cost per attribute therefore depends on the attributes already in the positional index, their distance to the attributes required for the current query, the cost of applying the data conversion function, etc. For other formats, the cost is actually fixed and easy to model, e.g., a file format that stores sequences of compressed tuples will have a constant time to access each tuple (i.e., the decompression cost per tuple). To overcome the complexity stemming from format heterogeneity, ViDa uses a wrapper per file format, similar to Garlic [45]. When a query targets a specific file, the optimizer invokes the appropriate wrapper, which takes into account any auxiliary structures present, and normalizes access costs for the attributes requested. For example, for a CSV file for which no positional index structures exist, the cost to retrieve a tuple might be estimated to be $3 \times const\_cost$, where $const\_cost$ would be the corresponding cost in a DBMS. ViDa's optimizer uses these estimates to decide the appropriate points in a query plan to access the raw files.

Avoiding Cache Pollution. To avoid "polluting" its caches, ViDa decides how eagerly to feed raw data into the query engine via the access paths. Identifying the attributes necessary for query evaluation and feeding them into ViDa's engine as the first step of answering a query is a straightforward option. Large, complex objects (e.g., JSON deep hierarchies) materialized as the result of a projected attribute of a query, however, will "pollute" ViDa's caches[1]. By carrying only the starting and ending *binary positions* of large objects through query evaluation, ViDa can avoid these unnecessary costs. ViDa retrieves these positions from its positional indexes, and exploits them at the time of result projection to assemble only the qualifying objects. Still, accessing raw files more than once during query evaluation may increase costs; ViDa thus takes into account trade-offs between I/O, memory and CPU costs. Figure 4 shows different potential data layouts for a given query that requires an integer field and a JSON object for its evaluation.

[1]Eagerly loading scalars alone already increases the creation cost of data structures significantly [31].

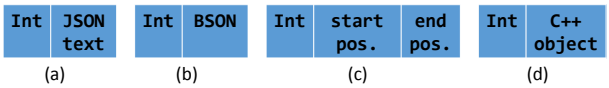

Figure 4: Potential layouts for a tuple containing a JSON object. The optimizer decides which one to output.

Re-using and re-shaping results. ViDa can keep copies of the same information of interest in its caches using different data layouts and use the most suitable layout during query evaluation of different workloads. As ViDa's engine allows for arbitrary operator inputs, it can cache replicas of tabular, row-oriented data in a columnar format and process them in a column-oriented manner for some queries. Similarly, ViDa can maintain both object-oriented and flattened versions of the same hierarchical datasets.

Queries in ViDa can also request the output in a custom layout, regardless of the original layout of the raw data, further affecting optimization decisions. If an application built on top of ViDa offers a JSON RESTful interface, ViDa materializes intermediate results as binary JSON to avoid reconstructing the objects for every query. For business applications that generate reports, existing intermediate results in a tabular layout can be served to users as additional reports, so that they may derive more insights from the data.

6. EXPERIMENTAL RESULTS

In this section, we present a preliminary evaluation analysis of the ViDa prototype in a real-life scenario.

ViDa Prototype. The language layer of ViDa offers the complete expressive power of monoid comprehensions. It supports queries over JSON, CSV, XLS, ROOT, and files containing binary arrays. The just-in-time executor as well as the optimizer of ViDa are ongoing work. In the following experiments, we use the prototype of the just-in-time executor for queries over CSV and JSON that exclusively access either ViDa's caches or the underlying data files. For the rest of the queries in the experiments, which have to access both the raw files and the caches of ViDa, we use a static pre-generated executor for which we had implemented more sophisticated caching mechanisms.

To efficiently navigate through the raw CSV and JSON files, ViDa uses auxiliary positional indexes, which capture structural information of the raw data files. For CSV files, "positional maps" [3] store the byte positions of attributes in the raw data files. For JSON files, we similarly maintain positional information such as starting and ending positions of JSON objects and arrays [43]. ViDa also caches previously accessed data fields.

The upper layers of ViDa (parser, rewriter, optimizer, etc.) are written in Scala, which eases the expression of the optimizer's rewrite rules. The just-in-time executor is being written in C++, and uses the LLVM compiler infrastructure [36] to generate code at runtime. The static executor is written in GO, exploiting GO's channels to offer pipelined execution. The access paths and the auxiliary structures of ViDa are written in C++.

Experimental Methodology & Setup. We compare ViDa with two popular approaches for integrating data stored

Relation name	Tuples	Attributes	Size	Type
Patients	41718	156	29 MB	CSV
Genetics	51858	17832	1.8 GB	CSV
BrainRegions	17000	20446	5.3 GB	JSON

Table 2: Human Brain Project - Workload characteristics.

in heterogeneous formats, namely i) integrating all files in one data warehouse and ii) employing different systems to accommodate files of different formats (e.g., a document-oriented DBMS for JSON files and an RDBMS for CSV files). This demonstration aims to highlight the trade-offs of the different approaches.

We use two state-of-the-art open-source relational database systems, a column-store (MonetDB) and a row-store (PostgreSQL), for storing relational data. We use MongoDB for storing hierarchical data. To integrate files of different formats in one data warehouse, we perform the data transformations into a common format (e.g., normalize/flatten a JSON file to CSV) and we then load the data in the RDBMS. When different systems are used, a data integration layer on top of the existing systems (the RDBMS of choice and MongoDB) is responsible for providing access to the data.

All experiments are conducted in a Sandy Bridge server with a dual socket Intel(R) Xeon(R) CPU E5-2660 (8 cores per socket @ 2.20 GHz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 20 MB L3 cache shared, and 128 GB RAM running Red Hat Enterprise Linux 6.3 (Santiago - 64bit). The server is equipped with a RAID-0 of 7 250 GB 7500 RPM SATA disks.

Workload and Datasets. For our experiments, we use a query workload and datasets from a real-life application of the Human Brain project (see Section 1.1). We use as input two relations (*Patients* and *Genetics*) describing tabular data, stored in CSV files, and a hierarchical dataset (*BrainRegions*) which is stored in a JSON file. The relation *Patients* contains patient-related information (e.g., protein levels in a blood sample) and the relation *Genetics* describes DNA sequence variations ("SNPs") observed in a patient's DNA. Finally, the JSON file *BrainRegions* comprises a hierarchy of 17000 objects containing results of a processing algorithm run over an input of Magnetic resonance imaging (MRI) scans. Table 2 summarizes the workload characteristics. In the case of PostgreSQL, we vertically partition the input relations due to the limit on the number of attributes per table (250-1600 depending on attribute types). The ViDa prototype queries the raw datasets directly.

The input query workload is based on a typical analysis scenario in the context of medical informatics and features 150 queries. Such scenarios typically involve a sequence of between 100 and 200 queries for i) epidemiological exploration where datasets are filtered using geographical, demographic, and age criteria before computing aggregates over the results to locate areas of interest in the raw datasets, and ii) interactive analysis where the patient data of interest is joined with information from imaging file products (i.e., the JSON dataset). The results of the latter case can be visualized on a brain atlas (as they contain spatial information), or be used as input to a subsequent statistical analysis algorithm. Most queries access all three datasets, apply a number of filtering predicates, and project out 1-5 attributes.

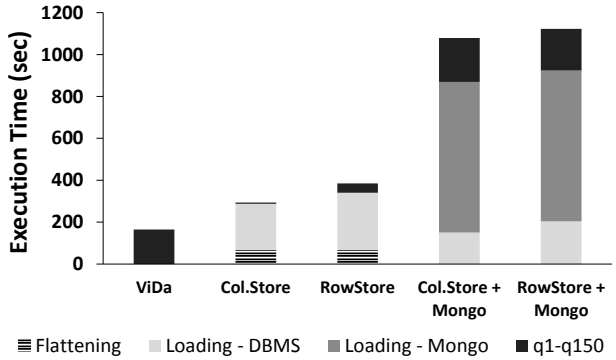

Figure 5: ViDa vs. i) an RDBMS containing flattened data, and ii) an RDBMS and a document DBMS, under an integration layer.

A query template describing them is:

```
for { p <- Patients, g <- Genetics, b <- BrainRegions,
      p.id = g.id, g.id = b.id, pred1, ..., predN
    } yield bag (attr1 := val1, ..., attrN := valN)
```

The equivalent SQL query is:

```
SELECT val1, ..., valN
FROM Patients p JOIN Genetics g ON (p.id = g.id)
JOIN BrainRegions b ON (g.id = b.id)
WHERE pred1 AND ... AND predN
```

Results. Figure 5 illustrates the cumulative time needed to prepare (i.e., flattening and data loading) and execute the sequence of queries with each system. "Col.Store" and "RowStore" correspond to the "single data warehouse" solution, using a column-store and a row-store respectively. Similarly, "Col.Store+Mongo" and "RowStore+Mongo" correspond to the case when different systems are used. We observe that the ViDa prototype achieves the best overall performance (up to 4.2x faster) and manages to compute the entire workload while the other approaches are still in the preparation phase. The preparation phase of the RDBMS-only solution includes data "flattening", which is both time consuming and introduces additional redundancy in the data stored. When different systems are used, the need for a data integration layer comes with a performance penalty during query processing. In addition, although no initial flattening was required, populating MongoDB was a time- but also a space-consuming process: the imported JSON data reached 12GB (twice the space of the raw JSON dataset).

The performance of individual queries in ViDa is comparable to that of a fully-loaded DBMS for most of the queries asked due to locality in the query workload. ViDa served approximately 80% of the workload using its data caches. For these queries, the execution time was comparable to that of the loaded column store in our experiments. For the remaining 20% of the query workload, queries had to access the raw data files, either because the raw files had not been accessed before, or because some additional fields (that were not cached by previous accesses) were needed to answer a query. Although queries that access multiple fields from the raw files in ViDa are more expensive than their counterparts running on data loaded in a DBMS, the majority of ViDa's cumulative execution time is actually spent in the initial accesses to the three datasets. This is a cost that both ViDa

and the DBMS-based approaches have to pay. Still, using faster storage would actually benefit ViDa, as some of the queries launched over it are I/O-bound, therefore ViDa's optimizations are masked by the I/O cost for these queries. For the other systems in the experiments, the loading and transformation process is CPU-intensive, therefore it is their CPU overheads that are partially masked by the initial I/O.

Summary. Our experiments show that the ViDa prototype offers performance that is competitive to state-of-the-art approaches for data analysis of heterogeneous datasets. ViDa serves a real-life query workload from the Human Brain Project before the competitive approaches finish data loading and transformation. At the same time, ViDa does not have to i) copy data, ii) "lock" data in a proprietary format, or iii) transform data. Data virtualization of heterogeneous raw data using ViDa is a viable option for the data analysis in this scenario.

7. DATA VIRTUALIZATION: NEXT STEPS

ViDa is an ongoing effort; we aim to examine numerous new opportunities enabled by data virtualization, which we will discuss in this section.

Complex, procedural analytics. Our focus in this work has been at providing support for a wide range of declarative queries, targeting a multitude of data formats. The monoid comprehension calculus, however, provides numerous constructs (e.g., variables, if-then-else clauses, etc.) that ViDa can already use to express tasks that would typically be expressed using a procedural language. The overall workload of a user can therefore benefit from the optimizer of ViDa, which has the potential to optimize tasks such as complex iterative machine learning in a holistic way; by translating both the data access and the computational parts to its intermediate algebra, ViDa can optimize them as a single procedure, and identify "cross-layer" optimization opportunities (similar to the ones in [46]). Procedural tasks typically involve state manipulation; monoids might appear as a counter-intuitive way to express such tasks because they are mostly used in the side-effect-free manner associated with functional programming. We can model state manipulation, however, either by introducing a *state transformation monoid* [21], or by introducing a global *context*, as Tupleware [14] does to enable its monadic algebraic operators to handle shared state.

Data Cleaning. Data curation is a significant step in the data analysis pipeline. In many cases, it becomes a manual, tedious process. ViDa can exploit its adaptive nature to reduce the effort required to clean input data sources. A conservative strategy starts by identifying entries whose ingestion triggers errors during the first access to raw data; then, the code generated for subsequent queries can explicitly skip processing of the problematic entries. Another opportunity comes from exploiting domain-specific knowledge for each data source. So far, ViDa has used domain-specific knowledge to optimize the generated code. Additional information about the domain, such as acceptable value ranges, dictionaries of values [48] (i.e., list of values that are valid for a given attribute), etc., can be incorporated in an input plugin that is specialized for that specific source. Then, different policies can be implemented for wrong values detected during scanning; options include skipping the invalid entry, or transforming it to the "nearest acceptable value" using a distance-based metric such as Hamming distance [25].

Integrating new storage technologies. Novel storage technologies are particularly important in data management because database systems' components have long been tuned with the characteristics of magnetic disk drives in mind. New storage technologies, however, have different characteristics. Flash, phase change memory (PCM) and memristor are the future in data center storage, while flash is currently the de facto storage medium for an increasing number of applications. ViDa must integrate new storage technologies carefully in the data virtualization architecture, considering the trade-off in cost, performance and energy consumption. We focus on i) new data placement strategies that avoid (slow) random writes on flash by modifying them into sequential writes, ii) using new storage technologies as a performance booster for data warehouses that are stored primarily on HDDs, and iii) offloading data to different storage technologies considering the power requirements and the number of accesses. Overall, our goal is to determine the most suitable storage device for the various tasks of raw data processing, such as raw data storage, temporary structures for query processing, and data caches storage.

Energy Awareness. Databases should use the underlying hardware in a judicious and energy-efficient way. Energy consumption is increasing significantly and it should be considered when designing large-scale data analysis frameworks Emerging hardware architectures require us to develop novel data structures and algorithms for efficient raw data processing. The goal is to develop task- and data-placement strategies that are simultaneously workload-, hardware- and data-aware. We must consider CPUs with different capabilities, such as low-energy chips vs server-class hardware, and devise placement strategies that offload work to the most adequate CPU given the task at hand: for instance, low-energy CPUs can handle positional structures and caching operations while others perform heavy-duty query processing or code generation. Regarding code generation, the code to be generated must take into account the capabilities of the platform on which it will be executed. Besides considering characteristics such as different cache sizes [35], low-energy CPUs might be more suitable as hosts for lightweight versions of the code which are not aggressively optimized. Protocol buffers [1], the serialization framework used by Google, makes similar distinctions.

8. RELATED WORK

ViDa draws inspiration and utilizes concepts from a wide range of research areas, from data exploration to functional programming languages. Influences and related work are presented in this section.

Data Exploration. As data sizes become larger and larger, data exploration has become necessary to quickly gain insight of vast amounts of data. Several researchers have recognized the challenges accruing from the big data explosion and the need to re-design traditional DBMS for the modern application scenarios both in business and sciences [3, 19, 27, 28, 29, 32, 40]. ViDa shares a similar vision and extends current approaches to bring together data originally stored in heterogeneous data stores. ViDa abstracts data and manipulates it regardless of its original format while adapting the query engine to the underlying data. By using virtualization and performing the query processing directly on raw data files, ViDa can provide novel opportunities for low-overhead data exploration.

In situ data analytics. Asking queries over raw data is a very active research area. The "NoDB philosophy" [3] advocates that in many scenarios database systems can treat raw data files as first-class citizens and operate directly over them. Parallel operators for in situ processing, taking into account system resources (CPU cycles and I/O bandwidth) were proposed by [12]. Data vaults [30] and the parallel system SDS/Q [6] ask queries directly over scientific file formats, emphasizing on array-based data. Multiple systems adhering to the MapReduce paradigm [16] are used to perform data analysis on data files stored in HDFS [26]. Systems such as Pig [42] and Hive [51] expose a declarative query language to launch queries, which are then transformed internally to MapReduce jobs. Another variation of systems operate over HDFS, but their internal query engine resembles a traditional parallel database [11, 34, 39, 52]. Finally, "invisible loading" [2] and Polybase [18] follow a hybrid approach; they use a Hadoop cluster and a DBMS to answer queries, transferring data between the two when needed.

ViDa adopts the "NoDB philosophy" and extends it by applying data virtualization to process data originally stored in heterogeneous data file formats. The ideas we describe can also be applied in the domain of "SQL-on-Hadoop" solutions presented above.

Code generation. The use of just-in-time code generation to answer database queries has re-gained popularity, years after its initial application in System R [10]. HIQUE [35] dynamically instantiates code templates to generate hardware-specific (e.g., cache-conscious) code. RAW [31] and H2O [4] use similar code generation mechanisms. RAW generates its access paths just-in-time to adapt to the underlying data files and to the incoming queries. H2O dynamically adapts its data storage layout based on the incoming query workload. HyPer [41], Impala [54], and Tupleware [14] employ more sophisticated code generation techniques, using the LLVM JIT compiler infrastructure [36]. The query engine of HyPer is push-based to simplify the control flow in the generated code and to exploit data locality via pipelining. Tupleware utilizes code compilation to speed up computationally intensive machine learning tasks. The JIT compiler of LLVM also allows for very fast compilation times, making query compilation costs almost insignificant. Another JIT compiler, the one of JVM, has also been used to generate code for queries [44]. On the other side of low-level code generation, LegoBase [33] advocates "abstraction without regret" and staged compilation; its query engine and its optimization rules are both written in the high-level language Scala. Different optimizations can be applied in every query translation step from the original Scala representation to the C code that is eventually generated. For ViDa, code generation is a valuable tool to handle the model heterogeneity and to generate internal structures and query operators suitable for the data and the query characteristics.

Query Comprehensions. List and monad comprehensions [9, 53] are popular constructs in (functional) programming languages. They have been used to iterate through collections in programming languages such as Haskell, Scala, F#, Python and JavaScript. From a database-oriented perspective, the Kleisli functional query system [55] uses the comprehension syntax and has been used as a facilitator for data integration tasks due to its expressive power [8]. RodentStore [15] uses list comprehensions as the basis for its storage algebra; it manipulates the physical representation of the data by utilizing the expressive nature of comprehensions to express transformations. LINQ [38] exposes query comprehension syntax and enables queries over numerous databases. Monoid comprehensions [22, 23] have also been proposed as the calculus for the translation of object-oriented query languages, offering a complete framework for the application of numerous optimization techniques [23, 24].

9. CONCLUSIONS

Existing data analysis solutions typically involve transforming all datasets in a single proprietary format and loading them in a warehouse prior to initiating analysis. Such solutions do not scale with the increasing volume and diversity of data and query workloads. In addition, they are incompatible with scenarios in which data movement is prohibitive, while they are not flexible enough for users to analyze their data ad-hoc.

In this paper, we made the case for ViDa, a novel system focusing on virtualizing data; abstracting data out of its form, and manipulating it regardless of the way it is stored or structured. ViDa handles the underlying model heterogeneity by using an expressive internal query language and employs code generation techniques to adapt its entire query engine to the underlying data and to the current query workload. Our preliminary results show that virtualization of heterogeneous raw data using ViDa is a viable option for data analysis, while being significantly more expressive and flexible than state-of-the-art solutions. ViDa's prototype implementation offers competitive performance without breaking the rigid requirements of our use case: no data was copied, moved or "locked" in a proprietary format, and no transformations were needed.

Acknowledgments. We would like to thank the reviewers for their valuable comments and suggestions on how to improve the paper. This work has been partially funded by the Swiss National Science Foundation, project No. CRSII2 136318/1, "Trustworthy Cloud Storage" and the following EU FP7 programme projects: "BigFoot - Big Data Analytics of Digital Footprints" (Grant No 317858), "ViDa: Transforming Raw Data into Information through Virtualization" (Grant No 617508), and the "Human Brain Project" (Grant No 604102).

10. REFERENCES

[1] Protocol Buffers: Developer Guide. `https://developers.google.com/protocol-buffers/docs/overview`.

[2] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT*, 2013.

[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.

[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, 1998.

[6] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *SIGMOD*, 2014.

[7] R. Brun and F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *AIHENP'96 Workshop*, 1997.

[8] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In *VLDB*, 1995.

[9] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. pages 87–96, 1994.

[10] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[11] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop. In *SIGMOD*, pages 1223–1234, 2014.

[12] Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD*, 2014.

[13] W. Corno, F. Corcoglioniti, I. Celino, and E. D. Valle. Exposing Heterogeneous Data Sources as SPARQL Endpoints through an Object-Oriented Abstraction. In *ASWC*, 2008.

[14] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, 2014.

[15] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.

[17] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[18] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *SIGMOD*, 2013.

[19] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD*, 2014.

[20] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann.

[21] L. Fegaras. Optimizing Queries with Object Updates. *J. Intell. Inf. Syst.*, 12(2-3):219–242, 1999.

[22] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *SIGMOD*, 1995.

[23] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[24] T. Grust and M. H. Scholl. Translating OQL into Monoid Comprehensions – Stuck with Nested Loops? Technical report, 1996.

[25] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26:147–160, 1950.

[26] HDFS. Architecture Guide. `http://hadoop.apache.org/docs/stable/hdfs_design.html`.

[27] T. Heinis, M. Branco, I. Alagiannis, R. Borovica, F. Tauheed, and A. Ailamaki. Challenges and Opportunities in Self-Managing Scientific Databases. *IEEE Data Eng. Bull.*, 34(4):44–52, 2011.

[28] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

[29] S. Idreos and E. Liarou. dbTouch: Analytics at your fingertips. In *CIDR*, 2013.

[30] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM*, 2012.

[31] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[32] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB*, 4(12):1474–1477, 2011.

[33] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[34] M. Kornacker and J. Erickson. Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real. `http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/`.

[35] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[36] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[37] H. Markram, K. Meier, T. Lippert, S. Grillner, R. S. Frackowiak, S. Dehaene, A. Knoll, H. Sompolinsky, K. Verstreken, J. DeFelipe, S. Grant, J. Changeux, and A. Saria. Introducing the Human Brain Project. In *Procedia CS*, volume 7, pages 39–42, 2011.

[38] E. Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, 2011.

[39] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[40] A. Nandi and H. V.Jagadish. Guided Interaction: Rethinking the Query-Result Paradigm. In *VLDB*, 2011.

[41] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9), 2011.

[42] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[43] G. Ottaviano and R. Grossi. Semi-Indexing Semi-Structured Data in Tiny Space. In *CIKM*, 2011.

[44] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, 2006.

[45] M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, 1997.

[46] X. Shi, B. Cui, G. Dobbie, and B. C. Ooi. Towards Unified Ad-hoc Data Processing. In *SIGMOD*, pages 1263–1274, 2014.

[47] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *SSDBM*, 2011.

[48] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data Curation at Scale: The Data Tamer System. In *CIDR*, 2013.

[49] The FITS Support Office. FITS. `http://fits.gsfc.nasa.gov/`.

[50] The HDF Group. HDF5. `http://www.hdfgroup.org/HDF5`.

[51] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[52] M. Traverso. Presto: Interacting with petabytes of data at Facebook. `https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920`.

[53] P. Wadler. Comprehending Monads. In *LISP and Functional Programming*, pages 61–78, 1990.

[54] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014.

[55] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.

[56] K. Q. Zhu, K. Fisher, and D. Walker. LearnPADS++ : Incremental Inference of Ad Hoc Data Formats. In *PADL*, 2012.